1 Linked Lists

Lab Objective: *Analyzing and manipulating data are essential skills in scientific computing. Storing, retrieving, and rearranging data take time. As a dataset grows, so does the amount of time it takes to access and analyze it. To write effecient algorithms involving large data sets, it is therefore essential to be able to design or choose the data structures that are most optimal for a particular problem. In this lab we begin our study of data structures by constructing a generic linked list, then using it to implement a few common data structures.*

Introduction

Data structures are specialized objects for organizing data efficiently. There are many kinds, each with specific strengths and weaknesses, and different applications require different structures for optimal performance. For example, some data structures take a long time to build, but once built their data are quickly accessible. Others are built quickly, but are not as efficiently accessible. These strengths and weaknesses are determined by how the structure is implemented.

Python has several built-in data structure classes, namely `list`, `set`, `dict`, and `tuple`. Being able to use these structures is important, but selecting the correct data structure to begin with is often what makes or breaks a good program. In this lab we create a structure that mimics the built-in list class, but that has a different underlying implementation. Thus our class will be better than a plain Python list for some tasks, but worse for others.

Nodes

Think of data as several types of objects that need to be stored in a warehouse. A *node* is like a standard size box that can hold all the different types of objects. For example, suppose a particular warehouse stores lamps of various sizes. Rather than trying to carefully stack lamps of different shapes on top of each other, it is preferable to first put them in boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier. A *data structure* is like the warehouse, which specifies where and how the different boxes are stored.

A node class is usually simple. The data in the node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure.

Problem 1. Consider the following generic node class.

```
class Node:
    """A basic node class for storing data."""
    def __init__(self, data):
        """Store 'data' in the 'value' attribute."""
        self.value = data
```

Modify the constructor so that it only accepts data of type `int`, `float`, or `str`. If another type of data is given, raise a `TypeError` with an appropriate error message. Modify the constructor docstring to document these restrictions.

NOTE

Often the data stored in a node is actually a *key* value. The key might be a memory address, a dictionary key, or the index of an array where the true desired information resides. For simplicity, in this and the following lab we store actual data in node objects, not references to data located elsewhere.

Linked Lists

A *linked list* is a data structure that chains nodes together. Every linked list needs a reference to the first node in the chain, called the `head`. A reference to the last node in the chain, called the `tail`, is also often included. Each node instance in the list stores a piece of data, plus at least one reference to another node in the list.

The nodes of a *singly linked list* have a single reference to the next node in the list (see Figure 1.1), while the nodes of a *doubly linked list* have two references: one for the previous node, and one for the next node in the list (see Figure 1.2). This allows for a doubly linked list to be traversed in both directions, whereas a singly linked list can only be traversed in one direction.

```
class LinkedListNode(Node):
    """A node class for doubly linked lists. Inherits from the 'Node' class.
    Contains references to the next and previous nodes in the linked list.
    """
    def __init__(self, data):
        """Store 'data' in the 'value' attribute and initialize
        attributes for the next and previous nodes in the list.
        """
        Node.__init__(self, data)       # Use inheritance to set self.value.
        self.next = None
        self.prev = None
```

Now we create a new class, `LinkedList`, that will link `LinkedListNode` instances together by modifying each node's `next` and `prev` attributes. The list is empty initially, so we assign the `head` and `tail` attributes the placeholder value `None`.

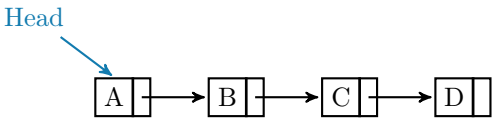

Figure 1.1: A singly linked list. Each node has a reference to the next node in the list. The head attribute is always assigned to the first node.

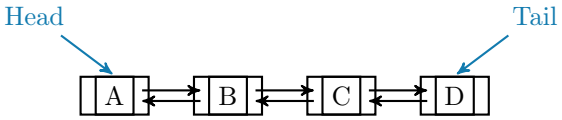

Figure 1.2: A doubly linked list. Each node has a reference to the node before it and a reference to the node after it. In addition to the head attribute, this list has a tail attribute that is always assigned to the last node.

We also need a method for adding data to the list. The `append()` makes a new node and adds it to the very end of the list. There are two cases to consider: appending to an empty list, and appending to a nonempty list. See Figure 1.3.

```
class LinkedList:
    """Doubly linked list data structure class.

    Attributes:
        head (LinkedListNode): the first node in the list.
        tail (LinkedListNode): the last node in the list.
    """
    def __init__(self):
        """Initialize the 'head' and 'tail' attributes by setting
        them to 'None', since the list is empty initially.
        """
        self.head = None
        self.tail = None

    def append(self, data):
        """Append a new node containing 'data' to the end of the list."""
        # Create a new node to store the input data.
        new_node = LinkedListNode(data)
        if self.head is None:
            # If the list is empty, assign the head and tail attributes to
            # new_node, since it becomes the first and last node in the list.
            self.head = new_node
            self.tail = new_node
        else:
            # If the list is not empty, place new_node after the tail.
            self.tail.next = new_node               # tail --> new_node
            new_node.prev = self.tail               # tail <-- new_node
            # Now the last node in the list is new_node, so reassign the tail.
            self.tail = new_node
```

Figure 1.3: Appending a new node to the end of a nonempty doubly linked list. The red arrows are the new connections. Note that the `tail` attribute is adjusted.

Achtung!

The `is` comparison operator is **not** the same as the `==` comparison operator. While `==` checks for numerical equality, `is` evaluates whether or not two objects are at the same location in memory.

```
# This evaluates to True since the numerical values are the same.
>>> 7 == 7.0
True

# 7 is an int and 7.0 is a float, so they cannot be stored at the same
# location in memory. Therefore 7 'is not' 7.0.
>>> 7 is 7.0
False
```

For numerical comparisons, always use `==`. When comparing to built-in Python constants such as `None`, `True`, `False`, or `NotImplemented`, use `is` instead.

find()

The `LinkedList` class only explicitly keeps track of the first and last nodes in the list via the `head` and `tail` attributes. To access any other node, we must use each successive node's `next` and `prev` attributes.

```
>>> my_list = LinkedList()
>>> my_list.append(2)
>>> my_list.append(4)
>>> my_list.append(6)

# To access each value, we use the 'head' attribute of the LinkedList
# and the 'next' and 'value' attributes of each node in the list.
>>> my_list.head.value
2
>>> my_list.head.next.value
4
>>> my_list.head.next.next.value
6
```

```
>>> my_list.head.next.next is my_list.tail
True
>>> my_list.tail.prev.prev is my_list.head
True
```

Problem 2. Add a method called `find(self, data)` to the `LinkedList` class that returns the first node in the list containing `data` (return the actual `LinkedListNode` object, not its `value`). If no such node exists, or if the list is empty, raise a `ValueError` with an appropriate error message.
(Hint: if `current` is assigned to one of the nodes the list, what does the following line do?)

```
current = current.next
```

Magic Methods

Endowing data structures with magic methods makes it much easier to use it intuitively. Consider, for example, how a Python list responds to built-in functions like `len()` and `print()`. At the bare minimum, we should give our linked list the same functionality.

Problem 3. Add magic methods to the `LinkedList` class so it behaves more like the built-in Python list.

1. Write the `__len__()` method so that the length of a `LinkedList` instance is equal to the number of nodes in the list. To accomplish this, consider adding an attribute that tracks the current size of the list. It should be updated every time a node is successfully added or removed.

2. Write the `__str__()` method so that when a `LinkedList` instance is printed, its output matches that of a Python list. Entries are separated by a comma and one space, and strings are surrounded by single quotes. Note the difference between the string representations of the following lists:

```
>>> num_list = [1, 2, 3]
>>> str_list = ['1', '2', '3']
>>> print(num_list)
[1, 2, 3]
>>> print(str_list)
['1', '2', '3']
```

remove()

In addition to adding new nodes to the end of a list, it is also useful to remove nodes and insert new nodes at specified locations. To delete a node, all references to the node must be removed. Then

Python will automatically delete the object, since there is no way for the user to access it. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to `None`.

```
class LinkedList:
    # ...
    def remove(self, data):
        """Attempt to remove the first node containing 'data'.
        This method incorrectly removes additional nodes.
        """
        # Find the target node and sever the links pointing to it.
        target = self.find(data)
        target.prev.next = None                    # -/-> target
        target.next.prev = None                    # target <-/-
```

Removing all references to the target node will delete the node (see Figure 1.4). However, the nodes before and after the target node are no longer linked.

```
>>> my_list = LinkedList()
>>> for i in range(10):
...     my_list.append(i)
...
>>> print(my_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> my_list.remove(4)              # Removing a node improperly results in
>>> print(my_list)                 # the rest of the chain being lost.
[0, 1, 2, 3]                       # Should be [0, 1, 2, 3, 5, 6, 7, 8, 9].
```

Figure 1.4: Naïve Removal for Doubly linked Lists. Deleting all references pointing to C deletes the node, but it also separates nodes A and B from node D.

This can be remedied by pointing the previous node's `next` attribute to the node after the deleted node, and similarly changing that node's `prev` attribute. Then there will be no reference to the removed node and it will be deleted, but the chain will still be connected.

Figure 1.5: Correct Removal for Doubly linked Lists. To avoid gaps in the chain, nodes B and D must be linked together.

Problem 4. Modify the `remove()` method given above so that it correctly removes the first node in the list containing the specified data. Account for the special cases of removing the first, last, or only node.

ACHTUNG!

Python keeps track of the variables in use and automatically deletes a variable if there is no access to it. In many other languages, leaving a reference to an object without explicitly deleting it could cause a serious memory leak. See `https://docs.python.org/2/library/gc.html` for more information on Python's auto-cleanup system.

insert()

Problem 5. Add a method called `insert(self, data, place)` to the `LinkedList` class that inserts a new node containing `data` immediately before the first node in the list containing `place`. Account for the special case of inserting before the first node.

See Figure 1.6 for an illustration. Note that since `insert()` places a new node before an existing node, it is not possible to use `insert()` to put a new node at the end of the list or in an empty list (use `append()` instead).

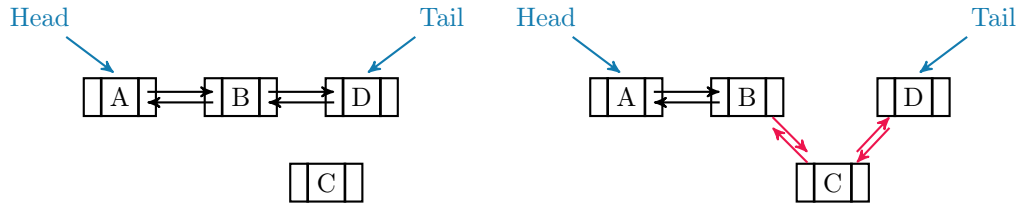

Figure 1.6: Insertion for Doubly linked Lists.

NOTE

The temporal complexity for inserting to the beginning or end of a linked list is $O(1)$, but inserting anywhere else is $O(n)$, where n is the number of nodes in the list. This is quite slow compared other data structures. In the next lab we turn our attention to *trees*, special kinds of linked lists that allow for much quicker sorting and data retrieval.

Restricted-Access Lists

It is sometimes wise to restrict the user's access to some of the data within a structure. The three most common and basic restricted-access structures are *stacks*, *queues*, and *deques*. Each structure restricts the user's access differently, making them ideal for different situations.

- **Stack**: *Last In, First Out* (LIFO). Only the last item that was inserted can be accessed. A stack is like a pile of plates: the last plate put on the pile is (or should be) the first one to be taken off. Stacks usually have two main methods: `push()`, to insert new data, and `pop()`, to remove and return the last piece of data inserted.
- **Queue** (pronounced "cue"): *First In, First Out* (FIFO). New nodes are added to the end of the queue, but an existing node can only be removed or accessed if it is at the front of the queue. A queue is like a line at the bank: the person at the front of the line is served next, while newcomers add themselves to the back of the line. Queues also usually have a `push()` and a `pop()` method, but `push()` inserts data to the end of the queue while `pop()` removes and returns the data at the front of the queue.[1]
- **Deque** (pronounced "deck"): a double-ended queue. Data can be inserted or removed from either end, but data in the middle is inaccessible. A deque is like a deck of cards, where only the top and bottom cards are readily accessible. A deque has two methods for insertion and two for removal, usually called `append()`, `appendleft()`, `pop()`, and `popleft()`.

Problem 6. Write a `Deque` class that inherits from the `LinkedList` class.

1. Use inheritance to implement the following methods:
   - `pop(self)`: Remove the last node in the list and return its data.
   - `popleft(self)`: Remove the first node in the list and return its data.
   - `appendleft(self, data)`: Insert a new node containing `data` at the beginning of the list.

   The `LinkedList` class already implements the `append()` method.
2. Override the `remove()` method with the following:

   ```
   def remove(*args, **kwargs):
       raise NotImplementedError("Use pop() or popleft() for removal")
   ```

   This effectively disables `remove()` for the `Deque` class, preventing the user from removing a node from the middle of the list.
3. Disable the `insert()` method as well.

Note

The `*args` argument allows the `remove()` method to receive any number of positional arguments without raising a `TypeError`, and the `**kwargs` argument allows it to receive any number of keyword arguments. This is the most general form of a function signature.

Python lists have `append()` and `pop()` methods, so they can be used as stacks. However, data access and removal from the front is much slower, as Python lists are not implemented as linked lists.

[1] `push()` and `pop()` for queues are sometimes called `enqueue()` and `dequeue()`, respectively)

The `collections` module in the standard library has a `deque` object, implemented as a doubly linked list. This is an excellent object to use in practice instead of a Python list when speed is of the essence and data only needs to be accessed from the ends of the list.

Problem 7. Write a function that accepts the name of a file to be read and a file to write to. Read the first file, adding each line to the end of a deque. After reading the entire file, pop each entry off of the end of the deque one at a time, writing the result to a line of the second file.

For example, if the file to be read has the list of words on the left, the resulting file should have the list of words on the right.

```
My homework is too hard for me.          I am a mathematician.
I do not believe that                    Programming is hard, but
I can solve these problems.              I can solve these problems.
Programming is hard, but                 I do not believe that
I am a mathematician.                    My homework is too hard for me.
```

You may use a Python list, your `Deque` class, or `collections.deque` for the deque. Test your function on the file `english.txt`, which contains a list of over 58,000 English words in alphabetical order.

Additional Material

Improvements to the Linked List Class

1. Add a keyword argument to the constructor so that if an iterable is input, each element of the iterable is immediately added to the list. This makes it possible to cast an iterable as a `LinkedList` the same way that an iterable can be cast as one of Python's standard data structures.

```
>>> my_list = [1, 2, 3, 4, 5]
>>> my_linked_list = LinkedList(my_list)  # Cast my_list as a LinkedList.
>>> print(my_linked_list)
[1, 2, 3, 4, 5]
```

2. Add new methods:
   - `count()`: return the number of occurrences of a specified value.
   - `reverse()`: reverse the ordering of the nodes (in place).
   - `rotate()`: rotate the nodes a given number of steps to the right (in place).
   - `sort()`: sort the nodes by their data (in place).
3. Implement more magic methods:
   - `__add__()`: concatenate two lists.
   - `__getitem__()` and `__setitem__()`: enable standard bracket indexing.
   - `__iter__()`: support `for` loop iteration, the `iter()` built-in function, and the `in` statement.

Other Linked List

The `LinkedList` class can also be used as the backbone for other data structures.

1. A *sorted list* adds new nodes strategically so that the data is always kept in order. A `SortedLinkedList` class that inherits from the `LinkedList` class should have a method called `add(self, data)` that inserts a new node containing `data` before the first node in the list that has a `value` that is greater or equal to `data` (thereby preserving the ordering). Other methods for adding nodes should be disabled.

   A linked list is **not** an ideal implementation for a sorted list (try sorting `english.txt`).
2. In a *circular linked list*, the "last" node connects back to the "first" node. Thus a reference to the tail is unnecessary.